C++学习笔记3

2022年3月19日 20:46

- 运算及其运算符
  - 重载运算符
    - C++语言定义了运算符作用于内置类型和复合类型运算对象时候所执行的操作。当运算符作用于类类型的运算对象时候，用户可以自行定义其含义
    - 使用重载运算符时，其包括运算对象的类型和返回值的类型都是由该运算符定义的；但是运算对象的个数，运算符的优先级和结合律都是无法改变的
  - 左值和右值
    - C++表达式要不然是右值，要不然就是左值
      - 注意
        - 一个左值表达式求值结果就是一个对象或者一个函数，然而以常量对象为代表的某些左值实际上是不能作为赋值语句的左侧运算对象
        - 此外，虽然某些表表达式求值结果是对象但是他们是右值而非左值
        - 只有在求值的时候才会出现左值和右值之说
      - 归纳
        - 当一个对象被用作右值的时候，用的是对象的值（内容）
        - 当对象被用作左值的时候，用的是对象的身份（在内存总的位置）
      - decltype推断的特殊性质
        - 如果表达式的求值结果是左值，decltype作用于该表达式（不是变量）得到一个引用类型
          - 假设p的类型是int*，因为解引用运算符生成左值，所以decltype（*p）的结果就是int&
          - 因为取地址运算符生成右值，所以decltype（&p）的结果就是int**，也就是说，结果是一个指向整型指针的指针
  - 处理复合表达式
    - 拿不准的时候最好用括号来强制让表达式的组合关系符合逻辑
    - 如果改变了某个运算对象的值，在表达式其他地方不要再使用这个运算对象
      - 例外
        - 当改变运算对象的子表达式本身就是另外一个子表达式的运算对象时候该规则无效（具体见4.1最后）
        -
  - sizeof运算符
    - sizeof运算符满足结合定律，其所得的值是size—t类型
    - 运算符对象的两种形式
      - sizeof expr
      - sizeof （type）
      - 其中type是数据类型，类似于int之类，expr是数据名称，类似于i，j等自定

义的数据名称

```
Sales_data data, *p;
sizeof(Sales_data);       // 存储 Sales_data 类型的对象所占的空间大小
sizeof data;              // data 的类型的大小，即 sizeof(Sales_data)
sizeof p;                 // 指针所占的空间大小
sizeof *p;                // p 所指类型的空间大小，即 sizeof(Sales_data)
sizeof data.revenue;      // Sales_data 的 revenue 成员对应类型的大小
sizeof Sales_data::revenue; // 另一种获取 revenue 大小的方式
```

- sizeof *p
  - 在这里无需担心*p是一个空指针，因为sizeof并不会实际去求解对象的值。
- sizeof几个特殊的地方
  - 对char或者类型为char的表达式执行sizeof运算，结果得1
  - 对引用类型执行sizeof运算得到被引用对象所占空间的大小
  - 对指针执行sizeof运算得到指针本身所占空间大小
  - 对解引用指针执行sizeof运算得到指针指向的对象所占空间的大小，指针不需要有效
  - 对数组执行sizeof运算得到整个数组所占空间大小，等价于对数组中所有元素各执行一次sizeof运算并将所得结果求和。
    - 注意：sizeof运算不会把数组转换成指针来处理
  - 对string对象和vector对象执行sizeof运算只返回该类型固定部分的大小，不会计算对象中的元素占用了多少空间

- 强制类型转换：建议避免

- 语句的注意事项
  - 在使用空语句的使用，有意停顿且加上注释让人们熟知
  - 在使用switch case语句的时候，
    - 可以多case联合使用

```
int main()
{
   unsigned int aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0;
   char ch;
   cout << "请输入一段文本：" << endl;
   while(cin >> ch)
   {
      switch(ch)
      {
       case 'a':
       case 'A':
          ++aCnt;
          break;
       case 'e':
       case 'E':
          ++eCnt;
          break;
       case 'i':
       case 'I':
          ++iCnt;
```


```
          break;
       case 'o':
       case 'O':
          ++oCnt;
          break;
       case 'u':
       case 'U':
          ++uCnt;
          break;
      }
   }
```

    - 注意case后面只能跟一个整型且是常量

- 异常处理机制
  - 异常处理机制为程序中异常检测和异常处理这两部分的协作提供支持
  - 异常处理包括
    - throw表达式
      - 异常检测部分使用throw表达式来表示它遇到了无法处理的问题。我们就说throw引发了异常
      - 程序的异常检测部分使用throw表达式引发一个异常，throw表达式包含关键字throw和紧随其后的一个表达式，其中表达式的类型就是抛出的异常类型。throw表达式后面通常紧跟一个分号，从而构成一条表达式语句
      - 例如：throw runtime_error("hello");
      - 注意：抛出异常 的时候会终止当前的函数，并且把控制权转移给能处理该异常的代码
    - try语句块：异常处理部分使用try语句块处理异常
      - try语句块以关键词try开始，并以一个或多个catch子句结束。try语句块中代码抛出的异常通常会被某个catch子句处理。因为catch子句“处理”异常，所以它们也被称作异常处理代码
      - 多try的嵌套
        - 类似于递归的方式处理代码
          - 当异常抛出的时候，首先搜索抛出该异常的函数。如果没找到匹配的catch子句，终止该函数，并在调用在该函数中继续寻找，如果还是没有找到匹配的catch语句，这个新的函数也会被终止，继续搜索调用它的函数，以此类推，沿着程序执行的路径逐层退回，直到找到适当类型的catch子句为止
          - 如果最终还是没能找到任何匹配的catch子句，程序转到名为termnate的标准库函数，该函数行为和系统有关，一般情况下，执行该函数将导致程序非正常退出
      - 注意
        - try语句块内的声明在变量的内部，在外部无法访问，特别是在catch子句中也无法访问
        - 出现一个try语句就代表着可能要抛出一个异常
      - 举例

```
try {
    program-statements
} catch (exception-declaration) {
    handler-statements
} catch (exception-declaration) {
    handler-statements
} // ...
```

    - 一套异常类

        - 用于在throw表达式和相关的catch子句之间传递异常的具体信息
    - 注意
        - 异常中断了程序的正常流程
            - 异常发生的时候，程序的一部分计算可能已经完成了，在这种情况下，略过部分程序意味着某些对象处理达到一半就戛然而止了，从而导致对象处于无效或者未完成的状态，或者资源没有正常释放
            - 哪些在异常发生期间正确执行了“清理”工作的程序被叫做异常安全的代码
            - 对于那些确实要处理异常并且继续执行的程序，就要加倍注意。我们必须时刻弄清楚异常何时发生，异常发生后程序应如何确保对象有效，资源无泄漏，程序处于合理状态等等
- 标准异常
    - 标准异常的几个头文件
        - exception头文件
            - 定义了通用的异常类exception。它只报告异常的的发生，不提供任何额外信息
        - stdexcept头文件
            - 定义了几种常用的异常类

表 5.1：<stdexcept>定义的异常类

exception	最常见的问题
runtime_error	只有在运行时才能检测出的问题
range_error	运行时错误：生成的结果超出了有意义的值域范围
overflow_error	运行时错误：计算上溢
underflow_error	运行时错误：计算下溢
logic_error	程序逻辑错误
domain_error	逻辑错误：参数对应的结果值不存在
invalid_argument	逻辑错误：无效参数
length_error	逻辑错误：试图创建一个超出该类型最大长度的对象
out_of_range	逻辑错误：使用一个超出有效范围的值

        - new头文件
            - new头文件中右bad_alloc异常类型
        - type_info头文件
            - 定义了bad_cast异常类型
    - 初始化
        - 对于exception，bad_alloc , bad_cast 对象：不能为这些对象提供初始值
        - 对于其他对象：使用string对象或者c风格字符串初始化，但是不允许使用默认初始化的方式
- 函数重点分析
    - 局部静态变量
        - 我们使用static 表示局部静态变量
        - 特点：一次初始化后，值就一致不会变了
    - 函数声明与分离式编译
        - 函数声明
            - 在头文件中进行函数声明

        - 函数的三要素描述了函数的接口，说明了调用该函数所需要的全部信息，函数声明也称作函数原型
        - 含有函数声明的头文件应该被包含到定义函数的源文件中
    - 分离式编译
        -
- 传指针和传引用
    - 熟悉C的程序员常常使用指针类型的形参访问函数外部对象。在C++语言中，建议使用引用类型的形参替代指针

```
//该函数接受一个 int 对象的引用，然后将对象的值置为 0
void reset(int &i)   // i 是传给 reset 函数的对象的另一个名字
{
    i = 0;           // 改变了 i 所引对象的值
}
```

    - 在上述调用过程中，i只是reset函数的对象的另外一个名字，在reset内部对i的使用就是对j的使用
    - 使用引用避免拷贝
        - 拷贝：指针，完全拷贝
        - 如果函数无需改变引用形参的值，最好将其声明为常量引用
    - 传入函数时候需要注意的点
        - 对于数字传递，传入函数时候会自动去更改数字的类型
        - string类型可以直接使用“abcdef”这样传入
    - const与函数
        - 对于传入值的函数，可以是const xx类型也可以不是const xx类型
            - 对于传引用/指针类型的函数，必须要将其一一进行类型对应
        - 如果不修改值的话，最好使用顶层const
- 数组与函数
    - 对于数组：不允许使用引用类型传递数组
    - 数组传递的三种形式
        - void print （const int*）
        - void print（const int 【】）
        - void print（const int 【10】）
    - 注意
        - 和其他数组一样,以数组作为对象的函数也必须确保使用数组的时候不会越界
    - 二维数组
        - 管理指针形参
            - 要求数组本身包含一个截止标记（使用标准库规范）
                - s。begin/s。end
            - 显式的传递一个表示数组大小的形参
                - const int （&/*arr）【10】：传一个数组，每个数组具有10个空间的二维数组
                - const int *arr【10】传一个指针的数组
            -

- 含有可变参数的函数
  - initializer—list形参
    - 类似于vector模板，也是可以放类型的
    - 也有迭代器，可以进行迭代器之类的操作
    - 和vector不一样的是，initializer_list的对象必须是常量值，我们永远无法改变initializer_list对象中元素的值
    - 操作类型
      - 操作

表 6.1：initializer_list 提供的操作

`initializer_list<T> lst;`	默认初始化；T 类型元素的空列表
`initializer_list<T> lst{a,b,c...};`	lst 的元素数量和初始值一样多；lst 的元素是对应初始值的副本；列表中的元素是 const
`lst2(lst)` `lst2 = lst`	拷贝或赋值一个 initializer_list 对象不会拷贝列表中的元素；拷贝后，原始列表和副本共享元素
`lst.size()`	列表中的元素数量
`lst.begin()`	返回指向 lst 中首元素的指针
`lst.end()`	返回指向 lst 中尾元素下一位置的指针

      - 初始化

```
initializer_list<string> ls; // initializer_list 的元素类型是 string
initializer_list<int> li;    // initializer_list 的元素类型是 int
```

      - 传入函数

```
// expected 和 actual 是 string 对象
if (expected != actual)
    error_msg({"functionX", expected, actual});
else
    error_msg({"functionX", "okay"});
```

    - 注意
      - 如果向里面传入一个值，必须要用花括号括起来
      - 传入的值可以不固定，但是必须都是常量
      -
  - 省略符形参
    - 省略符形参是为了便于C++程序访问某些特殊的C代码而设置的，这些代码使用了名为varargs的C标准库的功能。
    - 通常，省略符形参不应用于其他目的，你的C编译器文档会描述如何使用varargs
    - 注意
      - 省略符形参应该仅仅用于C和C++通用的类型，特别应该注意的是，大多类类型的对象在传递给省略符形参时都无法正确拷贝
      - 形式
- return语句
  - 注意
    - 在含有return语句的循环后面应该也有一条return语句，如果没有的话该程序就是错误的。
    - 返回局部对象的指针是错误的

- 返回局部对象的引用是错误的
- 对于引用类型的返回：注意只是返回的是return x，x的引用，且x必须是一个变量而不能是局部引用和局部指针

- 列表初始化返回值
  - 函数可以返回花括号包围值的列表
    - 类似于其他返回结果，此处的列表也用来对表示函数返回的临时量进行初始化。如果列表为空，临时量执行初始化；否则，返回的值由函数的返回类型决定
    - 如果函数返回的内置类型，则花括号包围的列表最多包含一个值（如int double等等）
    - 如果函数返回的是类类型，则由类本身定义初始值是如何使用的

- 声明返回数组指针/引用的函数
  - 方法1：类型别名
    - ```
      typedef int arrT[10];    // arrT 是一个类型别名，它表示的类型是含有 10 个
                               // 整数的数组
      using arrT = int[10];    // arrT 的等价声明，参见 2.5.1 节（第 60 页）
      arrT* func(int i);       // func 返回一个指向含有 10 个整数的数组的指针
      ```
  - 方法二：直接返回
    - 这种方法了解即可

      ```
      int (*func(int i))[10];
      ```

      可以按照以下的顺序来逐层理解该声明的含义：

      · func（int i）表示调用func函数时需要一个int类型的实参。

      · （* func（int i））意味着我们可以对函数调用的结果执行解引用操作。

      · （* func（int i））[10]表示解引用func的调用将得到一个大小是10的数组。

      · int （* func（int i））[10]表示数组中的元素是int类型。

  - 方法三：使用尾置返回类型
    - auto fun（）->int （*）【10】
      - 代码
      - 意思是：func函数返回了一个指针，这个指针指向含有十个整数的数组
    - decltype（fun1）* fun2（）
      ```

        - ◇ 代码
        - ◇ 意思是：fun1是一个数组，fun2的返回类型是一个指针，这个指针指向含有fun1那么多个数的数组

- 函数重载
    - 如果同一作用域内的几个函数名字相同但形参列表不同，我们称之为重载函数
    - 这些函数接受的形参的类型不一样，但是执行的操作非常相似。当调用这些函数时，编译器会根据传递的实参类型推断想要的是那个函数
    - 函数的名字仅仅是让编译器知道它调用的是哪个函数，而函数重载可以在一定程度上减轻程序员起名字，记名字的负担
    - 注意：main函数不能重载
    - 对于重载的函数来说，它们应该在形参的数量或者形参的类型上有所不同。
    - 不允许两个函数除了返回类型外其他所有的要素都相同
        - 注意：顶层const不会影响传入函数的对象。一个拥有顶层const的形参无法和另一个没有顶层const的形参区分开来，但是底层const可以
    - 注意
        - 除非两个函数操作非常相似 ，否则不要使用重载函数，否则会影响可读性
        - 调用重载函数的时候应该尽量避免强制类型转换。如果在实际应用中确实需要强制类型转换，则说明我们设计的形参类型不合理
    - 调用重载函数时候的可能
        - 编译器找到一个与实参最佳匹配的函数，并生成调用该函数的代码
        - 找不到任何一个函数与调用函数的实参匹配，此时编译器发出无匹配的错误信息
        - 有多于一个函数可以匹配，但是每一个都不是明显的最佳选择，此时也将发生错误，称为二义性调用
    - 重载与作用域
        - 一般来说，将函数声明置于局部作用域内不是一个明智的选择。
        - 如果我们在内层作用域中申明名字，它将隐藏外层作用域中声明的同名实体，在不同的作用域中无法重载函数名
        - 在C++语言中，名字查找发生在类型检查前
    - 特殊用途语言特性
        - 遵循尾部原则：添加和调用的时候，有默认实参函数的参数都要放在后面
        - 在声明处可以在函数的声明处赋值

```
int f(vector<int> v ,int i=2,int j=2);
```

        - 一旦有了默认形参，则函数后面再声明的时候只能对剩下的形参进行默认的声明，我们在后续的声明中是不能声明一个已经存在的默认值的
        - 有默认形参的声明一般都放在头文件中。
        - 由默认声明值的函数可以在使用的时候少放或不放值，不放值的地方就按照之前声明的默认值处理
        - 局部变量不能作为默认实参，除此以外，只要表达式的类型能转换成形参所需的类型，该表达式就能作为默认实参
            - □ 用作默认实参的名字再函数声明所在的作用域内解析，而这些名子求值的过

程发生在函数调用的时候

- 对于多个实参有默认声明的时候，不需要声明的实参需要空出来，只能省略尾部的实参

- 函数匹配
  - 谁最像和谁匹配，最像原则
  - 候选函数：所有的函数
  - 可行函数：可以带入值使用的函数
  - 寻找最佳匹配：精确匹配为好：完全一模一样带进去
  - 匹配的几种情况（最佳匹配，从上到下，越往上越精确）
    - 精确匹配
      - 实参类型和形参类型相同
      - 实参从对应的数组类型或函数类型转换到对应的指针类型
      - 向实参添加顶层const或者从实参中删除顶层const
    - 通过const转化实现的匹配
    - 通过类型提升实现的匹配
    - 通过算数类型转换或指针转换实现的匹配
    - 通过类类型转换实现的匹配
    - 注意
      - 内置类型提升和转换可能在函数匹配时候产生意想不到的效果。

- 内联函数和constexpr函数
  - 函数的好处
    - 阅读和理解函数的调用要比读懂等价的条件容易的多
    - 使用函数可以确保行为统一，每次相关操作都能保证按照同样的方式进行
    - 如果我们需要修改计算过程，显然修改函数要比先找到等价表达式所有出现的地方再逐一修改更容易
    - 函数可以被其他应用重复利用，省去了程序员重新编写代码的代价
  - 内联函数的好处
    - 调用内联函数比调用函数要快很多，内联函数的调用可避免函数调用的开销
  - 内联函数
    - 我们再函数之前加上inline关键字，这样就可以把他声明为内联函数
    - 内联函数一般都是return一个表达式，不要写的太多
    - 内联说明只是向编译器发出一个请求，编译器可以选择忽略这个请求
  - constexpr函数
    - constexpr函数是指能用于常量表达式的函数，定义时
      - 函数的返回类型及所有形参的类型都要是字面值类型
      - 函数体必须有且只有一条return语句
      - 为了能在编译的过程中随时展开，constexpr函数被隐式的指定为内联函数
    - constexpr函数不一定返回常量表达式
      - constexpr函数可以返回常量表达式也可以不返回常量表达式
      - 当传入是常量的时候，且运算结果为常量返回常量表达式

        - 当传入不是常量的时候，且运算结果非常量返回非常量表达式
- 调试宏
    - assert预处理宏
        - 表达式
            - assert（expr）
            - 首先对expr求值，如果表达式为假（0），assert输出信息并且终止程序的运行。如果表达式为真（非0）assert什么也不做
        - 定义头文件：cassert
            - 注意：因为assert为预处理器，因此我们可以直接使用预处理名字而无需提供using声明。
            - 在实际编程过程中，即使我们没有包含cassert头文件，也最好不要为了其他目的使用assert。很多头文件都包含了cassert，这就意味着即使你没有直接包含cassert，它也很有可能通过其他途径包括在你的程序中
    - NDEBUG预处理变量
        - assert的行为依赖于一个名为NDEBUG的预处理变量的状态，如果定义了NDEBUG，则assert什么也不做。默认状态下没有定义NDEBUG，此时assert将执行运行时检查
        - 我们可以使用一个#define语句定义NDEBUG，从而关闭调试状态
        - 错误调试中常用的四个名字
            - __FILE__存放文件名的字符串字面值
            - __LINE__存放当前行号的整型字面值
            - __TIME__存放文件编译时间的字符串字面值
            - __DATE__存放文件编译日期的字符串字面值
- 函数指针
    - 函数指针
        - 对于一个已经声明了的函数，要想声明一个可以指向该函数的指针，只需要用指针替换函数名即可
            - ```
              // pf 指向一个函数，该函数的参数是两个 const string 的引用，返回值是 bool 类型
              bool (*pf)(const string &, const string &); // 未初始化
              ```
            - 注意：
                - *pf两端的括号必不可少，如果不写这对括号，则pf是一个返回值为bool类型的指针的函数
    - 使用函数指针
        - 当我们把函数名作为一个值使用时，该函数将自动转化为指针
          ```
          pf = lengthCompare;          // pf 指向名为 lengthCompare 的函数
          pf = &lengthCompare;         // 等价的赋值语句：取地址符是可选的
          ```
        - 可以直接调用函数，调用函数时候可以直接写且无需解引用
          ```
          pf = lengthCompare;          // pf 指向名为 lengthCompare 的函数
          pf = &lengthCompare;         // 等价的赋值语句：取地址符是可选的


          bool b1 = pf("hello", "goodbye");         // 调用 lengthCompare 函数
          bool b2 = (*pf)("hello", "goodbye");      // 一个等价的调用
          bool b3 = lengthCompare("hello", "goodbye"); // 另一个等价的调用
          ```
        - 函数指针一旦指向了某个指针就不可以再次更改，除非对应的函数与原函数形参和
          ```

返回类型都匹配或者指向nullptr

```
string::size_type sumLength(const string&, const string&);
bool cstringCompare(const char*, const char*);
pf = 0;              // 正确：pf 不指向任何函数
pf = sumLength;      // 错误：返回类型不匹配
```

- 对于重载函数，要求极为精准的确切匹配

```
void ff(int*);
void ff(unsigned int);

void (*pf1)(unsigned int) = ff;  // pf1 指向 ff(unsigned)


void (*pf2)(int) = ff;       // 错误：没有任何一个 ff 与该形参列表匹配
double (*pf3)(int*) = ff;    // 错误：ff 和 pf3 的返回类型不匹配
```

- 声明另外一个函数的时候也可以同时声明一个函数的指针

```
// 第三个形参是函数类型，它会自动地转换成指向函数的指针
void useBigger(const string &s1, const string &s2,
               bool pf(const string &, const string &));
// 等价的声明：显式地将形参定义成指向函数的指针
void useBigger(const string &s1, const string &s2,
               bool (*pf)(const string &, const string &));
```

- 函数指针也可以像一般指针一样直接传入，也可以直接传入函数名字

```
// 自动将函数 lengthCompare 转换成指向该函数的指针
useBigger(s1, s2, lengthCompare);
```

- 使用typedef， using，decltype重新定义类型
  - typedef和using可以将函数/函数指针换一个名字

```
using F = int(int*, int);         // F 是函数类型，不是指针
using PF = int(*)(int*, int);     // PF 是指针类型
```

  - 使用decltype也可以重新命名，但是要注意decltype之后是一个函数，自己要加上*

```
string::size_type sumLength(const string&, const string&);
string::size_type largerLength(const string&, const string&);
// 根据其形参的取值，getFcn 函数返回指向 sumLength 或者 largerLength 的指针
decltype(sumLength) *getFcn(const string &);
```

- 总结
  - 函数指针和其他类型指针一样，只是在声明上有差别
  - 在声明的时候，为了返回不复杂，才映入了decltype，auto进行返回，不复杂
  - 对函数指针进行声明的时候，一定要记住，函数指针是指向返回类型相同，形参类型相同的哪一类，具体指向哪里要再次说明